CMPSC 465 Data Structures & Algorithms
Spring 2024 Mehrdad Mahdavi and David Koslicki

HW 10

1. (25 pts.) **Paths Dynamic Program.**
Subproblems:

Let $C(v)$ be the number of distinct paths from s to v.

Recursion:

The idea is that, if $C(u) = k$, i.e., vertex u can be reached from s in k distinct paths, and there is an edge (u, v), then v can be reached from s by taking one of these k distinct paths followed by edge (u, v). Combining all in-edges of v gives all possible paths. Therefore, the recursion is as follows:

$C(v) = \sum_{u \in (u,v) \in E} C(u)$

In order to guarantee that when we process v, all vertices in $\{u | (u, v) \in E\}$ have been processed, we need to traverse vertices in the order of linearization. Therefore, the first step of the algorithm is to compute a linearization using DFS. The pseudo-code, which includes three steps, is as follows.

Initialization:

compute a linearization of G
$C(v) = 0$, for any $v \in V$
$C(s) = 1$

Iteration:

for each v in the order of above linearization
  $C(v) = \sum_{u \in (u,v) \in E} C(u)$
end for

Termination:

$C(v)$ gives the number of distinct paths from s to v

Running time:

The linearization step takes $O(|V| + |E|)$ time. The initialization of $C(v)$ takes $O(|V|)$ time. The iteration step essentially traverses each edge at most once, and therefore it takes $O(|V| + |E|)$ time. The total running time of the algorithm is $O(|V| + |E|)$.

2. (25 pts.) **Weighted Set Cover.** As in the unweighted case, we will use a greedy algorithm:

```
while(some element of (B) is not covered)
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

Now we will prove that if there is a solution of cost k, then the above greedy algorithm will find a solution with cost at most $k \log_e n$.

After t iterations of the algorithm, let n_t be the number of elements still not covered, and let k_t be the total weight of the sets the algorithm chose, so $n_0 = n$ and $k_0 = 0$. Since the remaining n_t elements are covered by a collection of sets with cost k, there must be some set S_i such that S_i covers at least $w_i n_t/k$ new elements. (This is easiest to see by contradiction: if every set S_i covers less than $w_i n_t/k$ elements, then any collection with total weight k will cover less than kn_t/k elements.) This means that the ratio is n_t/k. Since the greedy algorithm picks the set with the highest ratio, we have that the chosen ratio $r \geq n_t/k$ so $w_r r \geq w_r n_t/k$ so the greedy algorithm will pick a set with at least $w_i n_t/k$ new elements. Therefore the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t(k_{t+1} - k_t)}{k} = n_t(1 - (k_{t+1} - k_t)/k).$$

Now, we apply the fact that for any x, $1 - x \leq e^{-x}$, with equality iff $x = 0$:

$$n_{t+1} < n_t e^{-(k_{t+1}-k_t)/k}.$$

By induction, we find that for $t > 0$, $n_t < n_0 e^{-k_t/k}$. If we choose the smallest t such that $k_t \geq k \log_e n$, then, n_t is strictly less than 1, which means no elements remain to be covered after t steps. The only time we can choose a set with weight k is if it covers all remaining elements, and it is not possible to choose a set with weight more than k due to the relationship discussed in paragraph 2. Since $k_{t-1} < k \log_e n$ and we will never add a set of weight more than k, it follows that $k_t < k \log_e n + k = O(k \log n)$.

3. (25 pts.) **Horn Formula.**

a
- Set everything to false initially
- x must be set to true since we have the statement $T \Rightarrow x$
- y must be set to true since $x \Rightarrow y$
- w must be set to true since $(x \wedge y) \Rightarrow w$
- Not all negative clauses are satisfied at this point, so there is no satisfying assignment.

b
- Set everything to false initially
- z must be set to true since we have the statement $\Rightarrow z$.
- w must be set to true since $z \Rightarrow w$
- x and y need not be changed, as all our implications are satisfied.
- All negative clauses are now satisfied, so we've found our satisfying assignment.

4. (25 pts.) **Longest Common Substring.** *Algorithm:* For $0 \leq i \leq n$ and $0 \leq j \leq m$, define $L(i, j)$ to be the length of the longest common postfix of $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$. When $i = 0$, it indicates $x_1 x_2 \dots x_i$ is an empty string. Similarly, when $j = 0$, it indicates $y_1 y_2 \dots y_j$ is an empty string. The recursion is:

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(0,0) = 0$. Then, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(i,0) = 0$$
$$L(0, j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$. We can find the maximum by either tracking the maximum when solving all $L(i, j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no postfix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common postfix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the postfixes i.e. the ending parts. Thus, going through all possible combinations of $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i,j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

Alternative Solution:
Algorithm: For $1 \le i \le (n+1)$ and $1 \le j \le (m+1)$, define $L(i,j)$ to be the length of the longest common prefix of $x_ix_{i+1}\dots x_n$ and $y_jy_{j+1}\dots y_m$. When $i = n+1$, it indicates $x_ix_{i+1}\dots x_n$ is an empty string. Similarly, when $j = m+1$, it indicates $y_jy_{j+1}\dots y_m$ is an empty string. The recursion is:

$$L(i,j) = \begin{cases} L(i+1,j+1)+1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(n+1,m+1) = 0$. Then, for all $1 \le i \le n$ and $1 \le j \le m$:

$$L(i,m+1) = 0$$
$$L(n+1,j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i,j)$ over all $1 \le i \le n$ (starting from n going back to 1) and $1 \le j \le m$ (starting from m going back to 1). We can find the maximum by either tracking the maximum when solving all $L(i,j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no prefix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common prefix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the prefixes i.e. the beginning parts. Thus, going through all possible combinations of $x_ix_{i+1}\dots x_n$ and $y_jy_{j+1}\dots y_m$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i,j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

5. (0 pts.) **Acknowledgments.**

(a) I did not work in a group.
(b) I did not consult with anyone other than my group members.
(c) I did not consult any non-class materials.

Rubric:

Problem 1, 25 pts

5 points for subproblems
3 points for base case
7 points for recurrence
5 for runtime analysis
5 for explaining why the algorithm works

Problem 2, 25 pts

- 10 pts for giving the algorithm including:

```
while(some element of (B) is not covered
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

- 15 pts for proving the algorithm

Problem 3, 25 pts

(a) 12.5 pts part a
- 1.5 pts for initial
- 3 pts for x assignment
- 3 pts for y assignment
- 3 pts for w assignment
- 2 pts for conclusion

(b) 12.5 pts part b
- 1.5 pts for initial
- 3 pts for z assignment
- 3 pts for w assignment
- 3 pts for answering x and y
- 2 pts for conclusion

Problem 4, 25 pts

- 5 points: subproblems
- 5 points: recurrence relation
- 3 points: initialization
- 2 points: retrieving final answer from computed DP
- 5 points: proof of correctness
- 5 points: running time analysis